DRAFT – DO NOT ENTER

REMARKS

CLAIM 20 IS REJECTED UNDER THE FIRST PARAGRAPH OF 35 U.S.C. § 112

Claim 20 has been cancelled; and thus, the rejection of claim 20 under the first paragraph of 35 U.S.C. § 112 is moot.

CLAIMS 21 AND 22 ARE REJECTED UNDER THE SECOND PARAGRAPH OF 35 U.S.C. § 112

Claims 21 and 22 have been cancelled; and thus, the rejection of claims 21 and 22 under the second paragraph of 35 U.S.C. § 112 is moot.

CLAIMS 23-44 ARE REJECTED UNDER THE SECOND PARAGRAPH OF 35 U.S.C. § 112

Regarding the Examiner's assertion that "it is unclear what the components of the recited 'machine' are," Applicants note that claims 23-44 recite a "machine for computing" (emphasis added). Applicants position is that one having ordinary skill in the art would have no difficulty understanding the scope of the phrase "machine for computing," particularly when reasonably interpreted in light of the written description of the specification.[1]

Claim 23 has been amended to clarify that the claimed invention is directed to a machine performing a method.

As to the Examiner's assertion that "the steps" lacks antecedent basis, Applicants respectfully submit that the use of the phrase "the steps comprising" in a preamble of a claim is ubiquitous, and one having ordinary skill in the art would not consider that phrase to be indefinite within the meaning of 35 U.S.C. § 112.

CLAIM 20 IS REJECTED UNDER THE FIRST PARAGRAPH OF 35 U.S.C. § 112

Claim 20 has been cancelled; and thus, the rejection of claim 20 under the first paragraph of 35 U.S.C. § 112 is moot.

CLAIMS 1-44 ARE REJECTED UNDER 35 U.S.C. § 101

On pages 2-3 of the Office Action, the Examiner asserted that the claimed invention, as recited in claims 1-44, is directed to non-statutory subject matter. This rejection is respectfully traversed.

The Examiner has alleged that the claimed invention, as recited in claims 1-44 "cite mere mathematical equations with indefinite post-solution activity or results." The Examiner also cited M.P.E.P. § 2106(IV)(B)(1) for support. This particular section of the M.P.E.P. relied upon In re Shrader[2] for support of the notion that "a process consisting solely of mathematical operations, i.e., converting one set of numbers into another set of

---

[1] In re Okuzawa, 537 F.2d 545, 190 USPQ 464 (CCPA 1976); In re Royka, 490 F.2d 981, 180 USPQ 580 (CCPA 1974).

[2] 22 F.3d 290, 30 USPQ2d 1455 (Fed. Cir. 1994).

DRAFT -- DO NOT ENTER

numbers, does not manipulate appropriate subject matter and thus cannot constitute a statutory process."

In re Shrader improperly relied upon

Notwithstanding the Patent Office's reliance upon In re Shrader in the M.P.E.P., both the Patent Office and the Federal Circuit recognizes that the court in In re Shrader did not apply a proper analysis. As stated in M.P.E.P. § 2106(I), "[o]ffice personnel should no longer rely on the Freeman-Walter-Abele test to determine whether a claimed invention is directed to statutory subject matter." The court in In re Shrader, however, relied on the Freeman-Walter-Abele test. The failure of the court in In re Shrader to rely upon a proper standard is discussed in AT&T Corp. v. Excel Communications, Inc.,[3] which states:

> Similarly, the court in In re Schrader relied upon the Freeman - Walter - Abele test for its analysis of the method claim involved. The court found neither a physical transformation nor any physical step in the claimed process aside from the entering of data into a record. See 22 F.3d at 294, 30 USPQ2d at 1458. The Schrader court likened the data-recording step to that of data-gathering and held that the claim was properly rejected as failing to define patentable subject matter. See id. at 294, 296, 30 USPQ2d at 1458-59. The focus of the court in Schrader was not on whether the mathematical algorithm was applied in a practical manner since it ended its inquiry before looking to see if a useful, concrete, tangible result ensued. Thus, in light of our recent understanding of the issue, the Schrader court's analysis is as unhelpful as that of In re Grams.

Therefore, the Examiner cannot properly rely upon In re Schrader to support the rejection of claims 1-44 under 35 U.S.C. § 101.

An algorithm is patentable when applied in "useful" way

In State Street Bank and Trust Company v. Signature Financial Group, Inc.,[4] the court elaborated on the mathematical algorithm exception to patentable subject matter by stating:

> Unpatentable mathematical algorithms are identifiable by showing they are merely abstract ideas constituting disembodied concepts or truths that are not "useful." From a practical standpoint, this means that to be patentable an algorithm must be applied in a "useful" way.

The court in State Street then set forth the criteria for establishing statutory subject matter under 35 U.S.C. § 101 as follows:

> The question of whether a claim encompasses statutory subject matter should not focus on which of the four categories of subject matter a claim is directed to — process, machine, manufacture, or composition of matter—but rather on the essential characteristics of the subject matter, in particular, its practical utility. Section 101 specifies that statutory subject matter must also satisfy the other "conditions and requirements" of Title 35, including novelty, nonobviousness, and adequacy of disclosure and notice. See In re Warmerdam, 33 F.3d 1354, 1359, 31 USPQ2d 1754, 1757-58 (Fed. Cir. 1994). For purpose of our analysis, as noted above, claim 1 is directed to a machine programmed with the Hub and Spoke software and admittedly produces a "useful, concrete, and tangible result." Alappat, 33 F.3d at 1544, 31 USPQ2d at 1557. This

[3] 172 F.3d 1352, 50 USPQ2d 1447 (Fed. Cir. 1999).
[4] 149 F.3d 1368, 47 USPQ2d 1596 (Fed Cir. 1998).

DRAFT -- DO NOT ENTER

> renders it statutory subject matter, even if the useful result is expressed in numbers, such as price, profit, percentage, cost, or loss.

Thus, as articulated above, the test for determining whether subject matter is patentable under 35 U.S.C. § 101 involves deciding if the subject matter produces a "useful, concrete, and tangible result." Furthermore, the law states that this result can be "expressed in numbers."

Applicants have established utility

A discussion of the procedural considerations regarding a rejection based upon lack of utility (i.e., 35 U.S.C. § 101) is found in M.P.E.P. § 2107.02. Specifically, M.P.E.P. § 2107.02(I) states that:

> regardless of the category of invention that is claimed (e.g., product or process), an applicant need only make one credible assertion of specific utility for the claimed invention to satisfy 35 U.S.C. 101 and 35 U.S.C. 112

In the paragraph spanning pages 1 and 2 of the disclosure and within the "Background of the Present Invention" section, Applicants stated the following:

> Steps for computing binary representations of numbers can create an unacceptably large deviation between an computer binary representation and its theoretical numerical value due to successive rounding errors. This can be an intolerable situation when a higher degree of accuracy is required. Various methods can improve computation accuracy but they may require a significant increase in processing time and/or hardware.

As recognized by those skilled in the art a floating-point number is a digital representation of an arbitrary real number in a computer. As alluded to by Applicants in the above-reproduced passage, rounding errors[5] with floating point numbers can degrade computation accuracy. In the second full paragraph on page 16 of the disclosure, Applicants stated with regard to the invention the following:

> Optionally, accuracy can be further improved by choosing x_i such that $p(x_i)$ has extra accuracy beyond the precision of the floating-point number system of the computer.

Applicants, therefore, have asserted a credible utility (i.e., improving the precision of a floating-point number system in a computer).

As noted in M.P.E.P. § 2107.02(III)(A), the Court of Customs and Patent Appeals in In re Langer[6] stated the following:

> As a matter of Patent Office practice, a specification which contains a disclosure of utility which corresponds in scope to the subject matter sought to be patented must be taken as sufficient to satisfy the utility requirement of § 101 for the entire claimed subject matter unless there is a reason for one skilled in the art to question the objective truth of the statement of utility or its scope. (emphasis in original)

Since a credible utility is contained in Applicants' specification, the utility requirement of 35 U.S.C. § 101 (i.e., whether the invention produces a useful, concrete, and tangible

---

[5] E.g., 2/3 is not perfectly represented by .66666667.
[6] 503 F.2d 1380, USPQ 288 (CCPA 1974).

DRAFT -- DO NOT ENTER

result) has been met. Therefore, Applicants respectfully solicit withdrawal of the imposed rejection of claims 1-44 under 35 U.S.C. § 101.

Floating point

SEE PAGES 1-2

From Wikipedia, the free encyclopedia.

A **floating-point number** is a digital representation for a number in a certain subset of the rational numbers, and is often used to approximate an arbitrary real number on a computer. In particular, it represents an integer or fixed-point number (the **significand** or, informally, the **mantissa**) multiplied by a base (usually 2 in computers) to some integer power (the **exponent**). When the base is 2, it is the binary analogue of scientific notation (in base 10).

A *floating-point calculation* is an arithmetic calculation done with floating-point numbers and often involves some approximation or rounding because the result of an operation may not be exactly representable.

A floating-point number a can be represented by two numbers m and e, such that $a = m \times b^e$. In any such system we pick a base b (called the *base* of numeration, also the *radix*) and a precision p (how many digits to store). m (which is called the *significand* or, informally, *mantissa*) is a p digit number of the form ±d.ddd...ddd (each digit being an integer between 0 and $b-1$ inclusive). If the leading digit of m is non-zero then the number is said to be **normalized**. Some descriptions use a separate sign bit (s, which represents −1 or +1) and require m to be positive. e is called the *exponent*.

This scheme allows a large range of magnitudes to be represented within a given size of field, which is not possible in a fixed-point notation.

As an example, a floating-point number with four decimal digits ($b = 10$, $p = 4$) and an exponent range of ±4 could be used to represent 43210, 4.321, or 0.0004321, but would not have enough precision to represent 432.123 and 43212.3 (which would have to be rounded to 432.1 and 43210). Of course, in practice, the number of digits is usually larger than four.

In addition, floating-point representations often include the special values $+\infty$, $-\infty$ (positive and negative infinity), and NaN ('Not a Number'). Infinities are used when results are too large to be represented, and NaNs indicate an invalid operation or undefined result.

Contents

Usage in computing

While in the examples above the numbers are represented in the decimal system (that is the base of numeration, $b = 10$), computers usually do so in the binary system, which means that $b = 2$. In computers, floating-point numbers are sized by the number of bits used to store them. This size is usually 32 bits or 64 bits, often called "single-precision" and "double-precision". A few machines offer larger sizes; Intel FPUs such as the Intel 8087 (and its descendants integrated into the x86 architecture) offer 80 bit floating point numbers for intermediate results, and several systems offer 128 bit floating-point, generally implemented in software. This website (http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html) can be used to calculate the floating point representation of a decimal number.

Problems with floating-point

Floating-point numbers usually behave very similarly to the real numbers they are used to approximate. However, this can easily lead programmers into over-confidently ignoring the need for numerical analysis. There are many cases where floating-point numbers do not model real numbers well, even in simple cases such as representing the decimal fraction 0.1, which cannot be exactly represented in any binary floating-point format. For this reason, financial software tends not to use a binary floating-point number representation. See:

Errors in floating-point computation can include:

- Rounding
  - Non-representable numbers: for example, the literal 0.1 cannot be represented exactly by a binary floating-point number
  - Rounding of arithmetic operations: for example 2/3 might yield 0.6666667
- Absorption: $1\times10^{15} + 1 = 1\times10^{15}$
- Cancellation: subtraction between nearly equivalent operands
- Overflow, which usually yields an infinity
- Underflow (often defined as an inexact tiny result outside the range of the normal numbers for a format), which yields zero, a subnormal number, or the smallest normal number
- Invalid operations (such as an attempt to calculate the square root of a non-zero negative number). Invalid operations yield a result of NaN (not a number).
- Rounding errors: unlike the fixed-point counterpart, the application of dither in a floating point environment is nearly impossible. See external references for more information about the difficulty of applying dither and the rounding error problems in floating point systems

Floating point representation is more likely to be appropriate when proportional accuracy over a range of scales is needed. When fixed accuracy is required, fixed point is usually a better choice.

Properties of floating point arithmetic

Arithmetic using the floating point number system has two important properties that differ from those of arithmetic using real numbers.

Floating point arithmetic is not associative. This means that in general for floating point numbers x, y, and z:

- $(x + y) + z \neq x + (y + z)$
- $(x \cdot y) \cdot z \neq x \cdot (y \cdot z)$

Floating point arithmetic is also not distributive. This means that in general:

- $x \cdot (y + z) \neq (x \cdot y) + (x \cdot z)$

In short, the order in which operations are carried out can change the output of a floating point calculation. This is important in numerical analysis since two mathematically equivalent formulas may not produce the same numerical output, and one may be substantially more accurate than the other.

For example, with most floating-point implementations, `(1e100 - 1e100) + 1.0` will give the result `1.0`, whereas `(1e100 + 1.0) - 1e100` gives `0.0`.

IEEE standard

The IEEE has standardized the computer representation for binary floating-point numbers in IEEE 754. This standard is followed by almost all modern machines. Notable exceptions include IBM Mainframes, which have both hexadecimal and IEEE 754 data types, and Cray vector machines, where the T90 series had an IEEE version, but the SV1 still uses Cray floating-point format.

As of 2000, the IEEE 754 standard is currently under revision. See: IEEE 754r

Examples

- The value of Pi, $\pi = 3.1415926..._{10}$ decimal, which is equivalent to binary $11.001001000011111..._2$. When represented in a computer that allocates 17 bits for the significand, it will become $0.11001001000011111 \times 2^2$. Hence the floating-point representation would start with bits 01100100100001111 and end with bits 10 (which represent the exponent 2 in the binary system). The first zero indicates a positive number, the ending $10_2 = 2_{10}$.

- The value of $-0.375_{10} = -0.011_2$ or -0.11×2^{-1}. In two's complement notation, -1 is represented as 11111111 (assuming 8 bits are used in the exponent). In floating-point notation, the number would start with a 1 for the sign bit, followed by 110000... and then followed by 11111111 at the end, or 1110...011111111 (where ... are zeros).

Hidden bit

When using binary ($b = 2$), one bit, called the **hidden bit** or the **implied bit**, can be omitted if all numbers are required to be normalized. The leading digit (most significant bit) of the significand of a normalized binary floating-point number is always non-zero; in particular it is always 1. This means that this bit does not need to be stored explicitly, since for a normalized number it can be understood to be 1.

The IEEE 754 standard exploits this fact. Requiring all numbers to be normalized means that 0 cannot be represented; typically some special representation of zero is chosen. In the IEEE standard this special code also encompasses denormal numbers, which allow for gradual underflow. The normalized numbers are also known as the normal numbers.

Note

Although the examples in this article use a consistent system of floating-point notation, the notation is different from the IEEE standard. For example, in IEEE 754, the exponent is between the sign bit and the significand, not at the end of the number. Also the IEEE exponent uses a biased integer instead of a two's complement number. The reader should note that the examples serve the purpose of illustrating how floating-point numbers could be represented, but the actual bits shown in the article are different from those in a IEEE 754-compliant representation. The placement of the bits in the IEEE standard enables two floating-point numbers to be compared bitwise (*sans* sign bit) to yield a result without interpreting the actual values. The arbitrary system used in this article cannot do the same.

See also

- Fixed-point arithmetic
- Computable number
- IEEE Floating Point Standard
- IBM Floating Point Architecture
- FLOPS

References

- An edited reprint of the paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic (http://docs.sun.com/source/806-3568/ncg_goldberg.html)*, by David Goldberg, published in the March, 1991 issue of Computing Surveys.
- David Bindel's Annotated Bibliography (http://www.cs.berkeley.edu/~dbindel/class/cs279/dsb-bib.pdf) on computer support for scientific computation.

- Kahan, William and Darcy, Joseph (2001). How Java's floating-point hurts everyone everywhere. Retrieved Sep. 5, 2003 from http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.
- Introduction to Floating point calculations and IEEE 754 standard (http://www.geocities.com/SiliconValley/Pines/6639/docs/fp_summary.html) by Jamil Khatib

Categories: Data types | Computer arithmetic

SEE PAGES 3-5, 8-9

9.1 Floating Point

This section under major construction.

One distinguishing feature that separates traditional computer science from scientific computing is its use of discrete mathematics (0s and 1s) instead of continuous mathematics and calculus. Transitioning from integers to real numbers is more than a cosmetic change. Digital computers cannot represent all real numbers exactly, so we face new challenges when designing computer algorithms for real numbers. Now, in addition to analyzing the running time and memory footprint, we must be concerned with the "correctness" of the resulting solutions. This challenging problem is further exacerbated since many important scientific algorithms make additional approximations to accommodate a discrete computer. Just as we discovered that some discrete algorithms are inherently too slow (polynomial vs. exponential), we will see that some floating point algorithms are too inaccurate (stable vs. unstable). Sometimes this problem can be corrected by designing a more clever algorithm. With discrete problems, the difficulty is sometimes intrinsic (NP-completeness). With floating point problems, the difficulty may also be inherent (ill conditioning), e.g., accurate long-term weather prediction. To be an effective computational scientist, we must be able to classify our algorithms and problems accordingly.

Floating point. Some of the greatest achievements of the 20th century would not have been possible without the floating point capabilities of digital computers. Nevertheless, this subject is not well understood by most programmers and is a regular source of confusion. In a February, 1998 keynote address entitled *Extensions to Java for Numerical Computing* James Gosling asserted "95% of folks out there are completely clueless about floating-point." However, the main ideas behind floating point are not difficult, and we will demystify the confusion that plagues most novices.

Here are two articles on floating point precision: What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg and How Java's Floating-Point Hurts Everyone Everywhere co-authored by Turing award winner William Kahn. Here's Wikipedia's entry on Numerical analysis.

IEEE 754 binary floating point representation. First we will describe how floating point numbers are represented. Java uses a subset of the IEEE 754 binary floating point standard to represent floating point numbers and define the results of arithmetic operations. Most machines conform to this standard, although some languages (C, C++) do not guarantee that this is the case. A `float` is represented using 32 bits, and each possible combination of bits represents one real number. This means that only 2^{32} possible real numbers can be exactly represented, evenly though there are infinitely many real numbers between 0 and 1. The IEEE standard uses an internal representation similar to scientific notation, but in binary instead of base 10. This covers a range from ±1.40129846432481707e-45 to ±3.40282346638528860e+38. with 6 or 7 significant decimal digits, including including plus infinity, minus infinity, and NaN (not a number). The number contains a sign bit *s* (interpreted as plus or minus), 8 bits for the

exponent *e*, and 23 bits for the mantissa *M*. The decimal number is represented according to the following formula.

$$(-1)^s \times m \times 2^{(e - 127)}$$

- *Sign bit s (bit 31).* The most significant bit represents the sign of the number (1 for negative, 0 for positive).
- *Exponent field e (bits 30 - 23).* The next 8 bits represent the exponent. By convention the exponent is *biased* by 127. This means that to represent the binary exponent 5, we encode 127 + 5 = 132 in binary (10000101). To represent the binary exponent -5 we encode 127 - 5 = 122 in binary (01111010). It is an alternative to two's complement notation used by IEEE.
- *Mantissa m (bits 22 - 0).* The remaining 23 bits represent the mantissa, *normalized* to be between 0.5 and 1. This normalization is always possible by adjusting the binary exponent accordingly. Binary fractions work like decimal fractions: 0.1101 represents 1/2 + 1/4 + 1/16 = 13/16 = 0.825. Not every decimal number can be represented as a binary fraction. For example 1/10 = 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + ... In this case, the number 0.1 is approximated by the closest 23 bit binary fraction 0.00011001100110011001..: One further optimization is employed. Since the mantissa always begins with a 1, there is no need to explicitly store this *hidden bit*.

As an example the decimal number 0.085 is stored as 00111101101011100001010001111011.

```
0.085:
bits:     31   30-23            22-0
binary:    0 01111011 01011100001010001111011
decimal:   0    123            3019899
```

This exactly represents the number $2^{e-127} (1 + m / 2^{23}) = 2^{-4}(1 + 3019899/8388608) = 11408507/134217728 = 0.085000000894069671630859375$.

A `double` is similar to a `float` except that its internal representation uses 64 bits, an 11 bit exponent with a bias of 1023, and a 52 bit mantissa. This covers a range from ±4.94065645841246544e-324 to ±1.79769313486231570e+308 with 14 or 15 significant digits of accuracy.

Here is a decimal to IEEE converter. To get the IEEE 754 bit representation of a `double` variable x use `Double.doubleToLongBits(x)`. According to The Code Project, to get the smallest double precision number greater than x (assuming x is positive and finite) is `Double.longBitsToDouble (Double.doubleToLongBits(x) + 1)`.

Precision vs. accuracy. Precision = tightness of specification. Accuracy = correctness. Do not confuse precision with accuracy. 3.133333333 is an estimate of the mathematical constant π which is specified with 10 decimal digits of precision, but it only has two decimal digits of accuracy. As John von Neumann once said

"There's no sense in being precise when you don't even know what you're talking about." Java typically prints out floating point numbers with 16 or 17 decimal digits of precision, but do not blindly believe that this means there that many digits of accuracy! Calculators typically display 10 digits, but compute with 13 digits of precision. Kahan: the mirror for the Hubble space telescope was ground with great precision, but to the wrong specification. Hence, it was initially a great failure since it couldn't produce high resolution images as expected. However, it's precision enabled an astronaut to install a corrective lens to counter-balance the error. Currency calculations are often defined in terms of a give precision, e.g., Euro exchange rates must be quoted to 6 digits.

Roundoff error. Programming with floating point numbers can be a bewildering and perilous process for the uninitiated. Arithmetic with integers is exact, unless the answer is outside the range of integers that can be represented (overflow). In contrast, floating point arithmetic is not exact since some number require an infinite number of digits to be represented, e.g., the mathematical constants e and π and 1/3. However, most novice Java programmers are surprised to learn that 1/10 is not exactly representable either in the standard binary floating point. These roundoff errors can propagate through the calculation in non-intuitive ways. For example, the first code fragment below from FloatingPoint.java prints `not equal`, while the second one prints `equal`.

```
double x1 = 0.3;
double x2 = 0.1 + 0.1 + 0.1;
System.out.println("x1 = " + x1 + ", x2 = " + x2);
if (x1 == x2) System.out.println("equal");
else          System.out.println("not equal");

double z1 = 0.5;
double z2 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
if (z1 == z2) System.out.println("equal");
else          System.out.println("not equal");
```

Following code fragment prints out 0.9200000000000002 twice!

```
System.out.println(0.9200000000000002);
System.out.println(0.9200000000000001);
```

A less contrived example is the following code fragment whose intent is to compute the square root of c by iterating Newton's method. Mathematically, the sequence of iterates converges to $\sqrt{c}$ from above, so that $t^2 - c > 0$. However, a floating point number only has finitely many bits of accuracy, so eventually, we may expect t^2 to equal c exactly, up to machine precision.

```
double EPSILON = 0.0;
double t = c;
while (t*t - c > EPSILON)
    t = (c/t + t) / 2.0;
```

Indeed, for some values of c, the method works.

```
% java Sqrt 2          % java Sqrt 4    % java Sqrt 10
1.414213562373095     2.0             3.1622776601683
```

This might give us some confidence that our code fragment is correct. But a surprising thing happens when we try to compute the square root of 20. Our program gets stuck in an infinite loop! Moreover, when we print out the iterates for computing the square root of 2, we observe that the loop actually terminates, not because (t^2 == c), but rather because (t^2 < c). Even though in mathematics, this quantity is guaranteed to be nonnegative, this is not necessarily the case with floating point arithmetic. This type of error is called *roundoff error*. *Machine accuracy* is smallest number ε such that (1.0 + ε != ε). In Java, it is XYZ with `double` and XYZ with `float`. Changing the error tolerance ε to a small positive value helps, but does not fix the problem (see exercise XYZ).

Every time you perform an arithmetic operation, you introduce an additional error of at least ε. Kernighan and Plauger: "Floating point numbers are like piles of sand; every time you move one you lose a little sand and pick up a little dirt." If the errors occur at random, we might expect a cumulative error of sqrt(N) ε_m. However, if we are not vigilant in designing our numerical algorithms, these errors can propagate in very unfavorable and non-intuitive ways, leading to cumulative errors of N ε_m or worse.

We must be satisfied with approximating the square root. A reliable way to do the computation is to choose some error tolerance ε, say 1E-15, and try to find a value t such that |t - c/t| < ε t. We use *relative error* instead of *absolute error*; otherwise the program may go into an infinite loop (see exercise XYZ).

```
double EPSILON = 1E-15;
double t = c;
while (Math.abs(t*t - c) > c*EPSILON)
    t = (c/t + t) / 2.0;
```

Harmonic sum. Possibly motivate by trying to estimate Euler's constant γ = limit as n approaches infinity of $\gamma_n = H_n$ - ln n. The limit exists and is approximately 0.5772156649. Surprisingly, it is not even known whether γ is irrational. γ_{100} = 0.582207332 is accurate to only one decimal place; $\gamma_{1000000}$ = 0.577216164 is accurate to only five decimal places, assuming no roundoff error. Better formula: $\gamma \approx \gamma_n - 1/(2n) + 1/(12n^2)$. Accurate to 12 decimal places for n = 1 million.

Many numerical computations (e.g., integration or solutions to differential equations) involve summing up alot of small terms. The errors can accumulate. Program HarmonicSum.java computes 1/1 + 1/2 + ... + 1/N using single precision and double precision. With single precision, when N = 10,000, the sum is accurate to 5 decimal digits, when N = 1,000,000 it is accurate to only 3 decimal digits, when N = 10,000,000 it is accurate to only 2 decimal digits. In fact, once N reaches 10 million, the sum never increases. Although the Harmonic sum diverges to infinity, in floating point it converges to a finite number! This dispels the popular misconception that if you are solving a problem that requires only 4

or 5 decimal digits of precision, then you are safe using a type that stores 7. Indeed, accumulation of roundoff error can lead to serious problems.

Financial computing. We illustrate some examples of roundoff error that can ruin a financial calculation. Financial calculations involving dollars and cents involve base 10 arithmetic. The following examples demonstrate some of the perils of using a binary floating point system like IEEE 754.

Sales tax. Program Financial.java illustrates the danger. Calculate the 9% sales tax of a 50 cent phone call. Using IEEE 754, 1.09 * 50 = xxx. This results gets rounded down to 0.xx even though the exact answer is 0.xx, which the telephone company (by law) is required to round down to 0.xx (using Banker's rounding). In contrast, a 14% tax on a 75 cent phone call yields xxx. might get rounded up to x.xx even though (by law) the phone company must round down the number (using Banker's rounding) to x.xx. This difference in pennies might not seem significant, and you might hope that the effects cancel each other out in the long run. However, taken over hundreds of millions of transactions, this might result in millions of dollars. Reference: Superman III where Richard Prior discovers a flaw in the computer system and embezzles fractions of a penny from every business transaction.

For this reason, some programmers believe that you should always use integer types to store financial values instead of floating point types. The next example will show the perils of storing financial values using type `int`.

Compound interest. This example introduces you to the dangers of roundoff error. Suppose you invest $1000.00 at 5% interest, compounded daily. How much money will you end up with after 1 year? If the bank computes the value correctly, you should end up with $1051.27 since the exact formula is

$$a * (1 + r/n)^n$$

which leads to 1051.2674964674473.... Suppose, instead, that the bank stores your balance as an integer (measured in pennies). At the end of each day, the bank calculates your balance and multiplies it by 1.05 and rounds the result to the nearest penny. Then, you will end up with only $1051.10, and have been cheated out of 17 cents. Suppose instead the bank rounds *down* to the nearest penny at the end of each day. Now, you will end up with only $1049.40 and you will have been cheated out of $1.87. The error of not storing the fractions of a penny accumulate and can eventually become significant, and even fraudulent. Program CompoundInterest.java.

Instead of using integer or floating point types, you should use Java's `BigDecimal` library. This library has two main advantages. First, it can represent decimal numbers exactly. This prevents the sales tax issues of using binary floating point. Second, it can store numbers with an arbitrary amount of precision. This enables the programmer to control the degree to which roundoff error affects the computation.

Other source of error. In addition to roundoff error inherent when using floating point arithmetic, there are some other types of approximation errors that commonly arise in scientific applications.

- *Measurement error.* The data values used in the computation are not accurate. This arises from both practical (inaccurate or imprecise measuring instruments) and theoretical (Heisenberg uncertainty principle) considerations. We are primarily interested in models and solution methods whose answer is not highly sensitive to the initial data.
- *Discretization error.* Another source of inaccuracy results from discretizing continuous system, e.g., approximating a transcendental function by truncating its Taylor expansion, approximating an integral using a finite sum of rectangles, finite difference method for finding approximate solutions to differential equations, or estimating a continuous function over a lattice. Discretization error would still be present even if using exact arithmetic. It is often unavoidable, but we can reduce truncation error by using more refined discretization. Of course, this comes at the price of using more resources, whether it be memory or time. Discretization error often more important than roundoff error in practical applications.
- *Statistical error.* Not enough random samples.

Catastrophic cancellation. Devastating loss of precision when small numbers are computed from large numbers by addition or subtraction. For example if x and y agree in all but the last few digits, then if we compute z = x - y, then z may only have a few digits of accuracy. If we subsequently use z in a calculation, then the result may only have a few digits of accuracy.

As a first example, consider the following contrived code fragment from Catastrophic.java.

```
double x1 = 10.000000000000004;
double x2 = 10.000000000000000;
double y1 = 10.00000000000004;
double y2 = 10.00000000000000;
double z = (y1 - y2) / (x1 - x2);
System.out.println(z);
```

It is easy to compute the exact answer by hand as (4/10,000,000,000,000) / (4/100,000,000,000,000) = 10. However, Java prints out result 11.5. Even though all calculations are done using 15 digits of precision, the result suddenly has only 1 digit of accuracy. Now we consider a more subtle and pernicious consequence of catastrophic cancellation. Program Exponential.java computes e^x using the Taylor series

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots$$

This series converges uniformly and absolutely for all values of x. Nevertheless, for many negative values of x (e.g., -25), the program obtains no correct digits, no matter how many terms in the series are summed. For example when x = -25, the series converges in floating point to -7.129780403672078E-7 (a negative number!), but the true answer is 1.3887943864964021E-11. To see why, observe that term 24 in the sum is $25^{24}/24!$ and term 25 is $-25^{25}/25!$. In principle, they should exactly cancel each other

out. In practice, they cancel each other out catastrophically. The size of these terms ($5.7 * 10^9$) is 20 orders of magnitude greater than the true answer, and any error in the cancellation is magnified in the calculated answer. Fortunately, in this case, the problem is easily rectified (see Exercise XYZ).

Numerical analysis. Lloyd Trefethen (1992) "Numerical analysis is the study of algorithms for the problems of continuous mathematics."

Stability. A mathematical problem is *well-conditioned* if it solution changes by only a small amount when the input parameters changes by a small amount. An algorithm is *numerically stable* if the output of the algorithm changes by only a small amount when the input data changes by a small amount. Numerical stability captures how errors are propagated by the algorithm. Numerical analysis is the art and science of finding numerically stable algorithms to solve well-conditioned problems. Accuracy depends on the conditioning of problem and the stability of the algorithm. Inaccuracy can result from applying a stable algorithm to an ill-conditioned problem or an unstable algorithm to a well-conditioned problem. For a simple example, computing $f(x) = \exp(x)$ is a well conditioned problem since $f(x + \varepsilon) = \ldots$. One algorithm for computing $\exp(x)$ is via its Taylor series. Suppose we estimate $f(x) = \exp(x)$ using the first four terms of its Taylor approximation: $g(x) = 1 + x + x^2/2 + x^3/3!$. Then $f(1) = 2.718282$, $g(1) = 2.666667$. However, it is unstable since if $x < 0$ ..... A stable algorithm is to use the Taylor series if x is nonnegative, but if x is negative, compute e^{-x} using a Taylor series and take the reciprocal.

Ill conditioning. In these previous section, we say an example of a non-stable algorithm for computing exp(x). We also discovered an improved algorithm for the problem that was stable. Sometimes, we are not always so lucky. We say that a problem is *ill-conditioned* if there is no stable algorithm for solving it.

Addition, multiplication, exponentiation, and division of positive numbers are all well-conditioned problems. So is computing the roots of a quadratic equation. (See Exercise XYZ.) Subtraction is ill-conditioned. So is finding the roots of a general polynomial. The conditioning of the problem of solving $Ax = b$ depends on the matrix A.

Population dynamics. The Verhulst equation is a simplified model of population dynamics.

$$x_n = (R + 1)x_{n-1} - R\,(x_{n-1})^2$$

Program Verhulst.java reads in a command line parameter R and iterates the Verhulst equation for 100 iterations, starting with $x_0 = 0.5$. It orders the computation in four different, but mathematically equivalent, ways. All lead to significantly different results when R = 3, none of which is remotely correct (as can be verified using exact arithmetic in Maple). (Reference)

	(R+1)x-R(xx)	(R+1)x-(Rx)x	((R+1)-(Rx))x	x +
0:	0.5000000000	0.5000000000	0.5000000000	0.5(
10:	0.3846309658	0.3846309658	0.3846309658	0.3(
20:	0.4188950250	0.4188950250	0.4188950250	0.4:

30:	0.0463994725	0.0463994756	0.0463994787	0.0
40:	0.3201767255	0.3201840912	0.3201915468	0.3
50:	0.0675670955	0.0637469566	0.0599878799	0.0
60:	0.0011449576	0.2711150754	1.0005313342	1.2
70:	1.2967745569	1.3284619999	1.3296922488	0.1
80:	0.5530384607	0.8171627721	0.0219521770	0.0
90:	0.0948523432	0.1541841695	0.0483069438	1.2
100:	0.0000671271	0.1194574394	1.2564956763	1.0

When R > 2.57, this system exhibits chaotic behavior. The system itself is ill-conditioned, so there is no way to restructure the computation to iterate the system using floating point arithmetic. Although we cannot expect our floating point algorithms to correctly handle ill-conditioned problem, we can ask that they report back an error rangle associated with the solution so that at least we are alerted to potential problems. For example, when solving linear systems of equations (see section 9.5), we can compute something called a *condition number*: this quantity can be used to bound the error of the resulting solution.

Ill-conditioning is not just a theoretical possibility. Astrophysicists have determined that our Solar System is chaotic. The trajectory of Pluto's orbit is chaotic, as in the motion of the Jovian planets, Halley's comet, and asteroidal zone trajectories. Upon a close encounter with Venus, Mercury could be ejected from the Solar System in less than 3.5 billion year!

Calculating special functions. When learning calculus, we derive convergent Taylor series formulas for calculating exponential and trigometric functions. (e.g., exp(x), log(x), sin(x), cos(x), arctan (x), etc.). However, we must take great care when applying such formulas on digital computers. Not all special functions are built in to Java's Math library, so there are times when you must create your own. Give example, e.g., error function.

Numerical analysts have derived accurate, precise, and efficient algorithms for computing classic functions (e.g., hyperbolic trigometric functions, gamma function, beta function, error function, Bessel functions, Jacobian elliptic functions, spherical harmonics) that arise in scientific applications. We strongly recommend using these proven recipes instead of devising your own ad hoc routines.

Real-world numerical catastrophes. The examples we've discussed above are rather simplistic. However, these issues arise in real applications. When done incorrectly, disaster can quickly strike.

Ariane 5 rocket. Ariane 5 rocket exploded 40 seconds after being launched by European Space Agency. Maiden voyage after a decade and 7 billion dollars of research and development. Sensor reported acceleration that so was large that it caused an overflow in the part of the program responsible for recalibrating inertial guidance. 64-bit floating point number was converted to a 16-bit signed integer, but the number was larger than 32,767 and the conversion failed. Unanticipated overflow was caught by a general systems diagnostic and dumped debugging data into an area of memory used for guiding the rocket's motors. Control was switched to a backup computer, but this had the same data. This resulted in a drastic attempt to correct the nonexistent problem, which separated the motors from their mountings, leading to the end of Ariane 5.

Patriot missile accident. On February 25, 1991 an American Patriot missile failed to track and destroy an Iraqi Scud missile. Instead it hit an Army barracks, killing 26 people. The cause was later determined to be an inaccurate calculate caused by measuring time in tenth of a second. Couldn't represent 1/10 exactly since used 24 bit floating point. Software to fix problem arrived in Dhahran on February 26. Here is more information.

Intel FDIV Bug Error in Pentium hardwire floating point divide circuit. Discovered by Intel in July 1994, rediscovered and publicized by math professor in September 1994. Intel recall in December 1994 cost $300 million. Another floating point bug discovered in 1997.

Sinking of Sleipner oil rig. Sleipner A $700 million platform for producing oil and gas sprang a leak and sank in North Sea in August, 1991. Error In inaccurate finite element approximation underestimate shear stress by 47% Reference.

Vancouver stock exchange. Vancouver stock exchange index was undervalued by over 50% after 22 months of accumulated roundoff error. The obvious algorithm is to add up all the stock prices after Instead a "clever" analyst decided it would be more efficient to recompute the index by adding the net change of a stock after each trade. This computation was done using four decimal places and truncating (not rounding) the result to three. Reference

Lessons.

- Use `double` instead of `float` for accuracy.
- Use `float` only if you really need to conserve memory, and are aware of the associated risks with accuracy. Usually it doesn't make things faster, and occasionally makes things slower.
- Be careful of calculating the difference of two very similar values and using the result in a subsequent calculation.
- Be careful about adding two quantities of very different magnitudes.
- Be careful about repeating a slightly inaccurate computation many many times. For example, calculating the change in position of planets over time.
- Designing stable floating point algorithms is highly nontrivial. Use libraries when available.

Q + A

Q. Is there any direct way to check for overflow on integer types?

A. No. The integer types do not indicate overflow in any way. Integer divide and integer remainder throw exceptions when the denominator is zero.

Q. What happens if I enter a number that is too large, e.g., `1E400`?

A. Java returns the error message "floating point number too large."

Q. What about floating point types?

A. Operations that overflow evaluate to plus or minus infinity. Operations that underflows result in plus or minus zero. Operations